A modelling language for the effective design of *Java* annotations

Irene Córdoba
Technical University of Madrid
irene.cordoba.sanchez@alumnos.upm.es

Juan de Lara
Autonomous University of Madrid
Juan.deLara@uam.es

ABSTRACT

This paper describes a new modelling language for the effective design of Java annotations. Since their inclusion in the 5th edition of Java, annotations have grown from a useful tool for the addition of meta-data to play a central role in many popular software projects. Usually they are conceived as sets with dependency and integrity constraints within them; however, the native support provided by Java for expressing this design is very limited.

To overcome its deficiencies and make explicit the rich conceptual model which lies behind a set of annotations, we propose a domain-specific modelling language.

The proposal has been implemented as an Eclipse plug-in, including an editor and an integrated code generator that synthesises annotation processors. The language has been tested using a real set of annotations from the Java Persistence API (JPA). It has proven to cover a greater scope with respect to other related work in different shared areas of application.

1. INTRODUCTION

In 2004 annotations where added to the Java language as an answer to the huge amount of boilerplate code that many APIs, such as JavaBeans or JAX Web Services (JAX-WS), required [2]. Other authors explain the appearance of annotations in Java as a result of the increasingly growing tendency of including the meta-data associated with a program within the program itself instead of keeping it in separate files; as well as the pressure from other programming languages which already included similar features, like C# [15].

Since their introduction in the language, annotations have become a success and are widely used in many important projects within the software development scene. We find them in frameworks like Seam [4] and Spring [5], in the Object Relation Mapping of Hibernate [1], and also in proper Sun Java standards such as the aforementioned JAX-WS, JavaBeans, Enterprise JavaBeans and the Java Persistence API (JPA) [3].

However, despite this success, the native support that Java provides for their development is very poor. On the one hand, the syntax for defining annotations is rather unusual for an accustomed Java programmer: some Java constructions are reused for other purposes that absolutely differ from their usual semantics. On the other hand, annotations are rarely conceived in an isolated way; instead they are usually part of a set with dependencies and integrity constraints. Moreover, each annotation separately usually carries integrity constraints with respect to the elements it can be attached to. Currently there is no effective way in Java for making explicit the constraints underlying a set of annotations at design time. Instead, the usual path taken to overcome this deficiencies is to develop an extension to the Java compilerto ensure that such constraints are complied with.

As a first step towards the alleviation of this situation, we propose *Ann*, a Domain-Specific Language (DSL) [9] aiming to provide a more expressive and suitable syntactic support for the design of sets of annotations and their associated integrity constraints. We have developed an integrated development environment as an Eclipse plug-in. The environment includes a code generator to translate the design and constraints expressed using *Ann* into Java code, which can be fully integrated in projects in such language. *Ann* has been tested using a real set of annotations from JPA, demonstrating that it can capture a wide set of the constraints in its specification. More information and source code of the project is available at `http://irenecordoba.github.io/Ann`.

The rest of the paper is organised as follows: section 2 gives a more detailed overview on the current limitations of Java annotations; section 3 overviews our approach; section

4 describes the proposed DSL, *Ann*; section 6 details a real case study for *Ann*; section 7 compares our proposal with related work; and section 8 summarises the conclusions and future development.

2. JAVA ANNOTATIONS

To help understanding the current limitations of Java annotations, in this section we describe how they are defined in Java (subsection 2.1), and how is their correct use checked (subsection 2.2).

2.1 Defining Java annotations

Java annotations do not constitute a type of their own. Instead, they are defined as *special* interfaces. Listing 1 shows an example of the definition of a simple annotation (called an *annnotation type*).

```
1  package examples;
2
3  import java.lang.annotation.Target;
4  import java.lang.annotation.ElementType;
5
6  @Target(ElementType.TYPE)
7  public @interface Person {
8     String name() default "Mary";
9     int age() default 21;
10    float weight() default 52.3f;
11 }
```

Listing 1: Annotation `Person` defined in Java.

As it can be noticed, the special nature of annotations is pointed out by the `@` character before the `interface` keyword (line 7). The zero-argument methods inside the container (lines 8-10) are the *fields* (the parameters) of the annotation. The notation is cumbersome because Java is providing a syntax characteristic of one construction (a method) to specify another completely different (an annotation parameter). Moreover, to assign a default value to those *fields*, the keyword `default` must be used, instead of the equality symbol, more natural and common in this context.

Finally, line 6 shows an example of an annotation being used: `Target`. This annotation is used to specify what kind of elements the declared annotation can annotate (*targets*). Although it is a way to introduce constraints in the definition of an annotation, it is very limited. For example, in this case by using the value `TYPE` of the enumeration `ElementType`, Person can only be applied to classes, interfaces (including annotation types) and enumerations. However, there is no way to e.g., to restrict its applicability to classes only.

This was a simple example, but if we take a look at the JPA documentation, we find that the annotation `Entity` can only be applied to classes meeting the following more elaborated requirements [3]:

- They must have a public or protected constructor.
- They must not be final.
- They must not have any final method.
- Their persistent fields must be declared private, protected or package-private.

None of these statements can be expressed nowadays with the syntax available for the definition of annotations.

What is more, when designing annotation sets, it is common to have constraints involving several annotations. For example, the JPA annotation `Id` is only allowed in attributes within classes annotated with `Entity`. We call such constraints the static semantics or integrity constraints of an annotation (set).

Therefore, what can be done to ensure the compliance of such outlined constraints? The only remaining choices are to write a guiding comment for its use and signal an error at runtime. In addition, it is possible to develop extensions to the Java compiler, known as annotation processors, an option we will detail more in the following subsection.

2.2 Annotation processors

The Java package `javax.annotation.processing` provides a set of elements for processing annotations at compile time. An annotation processor is invoked by the compiler, and it can check the annotations attached to any program element, performing an arbitrary task. Typically, the processor will check the correctness of the annotation placement (i.e., its static semantics), and may perform further actions (e.g., generating code). Annotation processing works in rounds. In each round a processor may be required to process a subset of the annotations found in the source code and the binary files produced in the prior round. If a processor was executed in a given round, it will be called again in the next rounds.

Listing 2 shows the structure of a typical annotation processor. Line 1 specifies the annotation to be checked, `Person` in this case. The key method of the processor is `process` (lines 5-23), where the elements annotated with the particular annotation are looked up and checked. If any of them does not satisfy the checks, then an error is raised using the functionality provided by the processing package (lines 15-20).

```
1  @SupportedAnnotationTypes("Person") // annotation to be checked
2  @SupportedSourceVersion(SourceVersion.RELEASE_6)
3  public class PersonProcessor extends AbstractProcessor
4  {
5     @Override
6     public boolean process(Set<? extends TypeElement> annotations,
7                            RoundEnvironment objects)
8     {
9        // iterate on all objects to check
10       for (Element elt: objects.getElementsAnnotatedWith(Person.class))
11       {
12          // evaluate correct placement of Person annotation for elt
13          ...
14          // if error
15          this.processingEnv.getMessager().printMessage
16          (
17             Kind.ERROR,
18             "The annotation @Person is disallowed for this location.",
19             elt
20          );
21       }
22       return true;
23    }
24 }
```

Listing 2: Structure of an annotation processor.

It is important not to confuse annotation processing with reflection. While the former takes place at compile time, the latter is at execution time. The values of an annotation at a given program element can be checked at execution time via the Java Reflection API, but it has several disadvantages, like an overhead in performance, the requirement of runtime permission (which may not be granted), and the possibility of breaking object-oriented abstractions.

In the context of checking the correctness of annotations, it is more appropriate to do it via annotation processors, be-

cause they can find and signal the errors without the need to execute the program. However, coding such processors is tedious and error prone. Moreover, we believe it would be advantageous to make explicit the underlying annotation constraints at a higher level, together with the annotation structure. For this purpose, we have created *Ann*, a DSL to define the structure and integrity constraints of Java annotations.

3. OVERVIEW OF THE APPROACH

Figure 1 shows the working scheme of our approach to solve the problems outlined in section 2. The main idea is to use a DSL [17] called Ann, to describe the syntax and static semantics of the family of annotations to be built in a declarative way (label 1 in the scheme). This DSL provides appropriate primitives for this task, beyond those natively offered by Java.

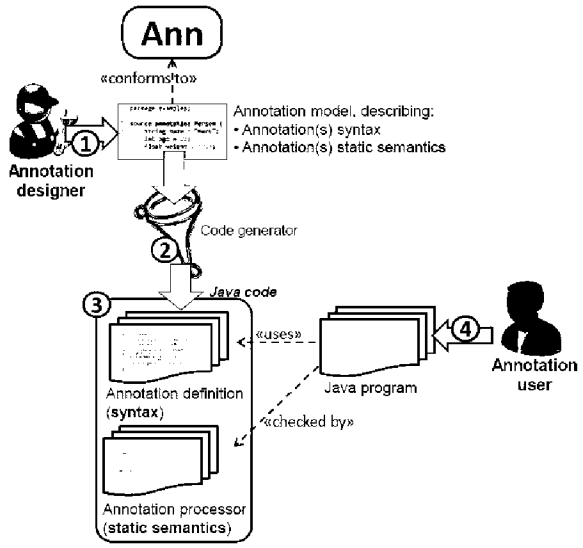

Figure 1: Overview of our approach.

Our solution includes a code generator (label 2) that produces plain Java files with the syntax definition and the annotation processors for the defined annotation (label 3). Then, the annotations can be safely used, because their correct use in Java programs is checked by the generated annotation processors.

Altogether, using Ann has several advantages, including: (i) it allows to make explicit the structure and integrity constraints of a set of annotations in a high-level, declarative way; and (ii) it automatically produces the annotation processors to check the correct use of annotations.

The next section details the elements of the Ann language.

4. THE ANN DSL

As we have previously stated, Ann is a domain-specific modelling language aimed at the description of the syntax and static semantics of Java annotations. Modelling languages are conceptual tools for describing reality explicitly, from a certain level of abstraction and under a certain point of view [9]. They are defined by three key elements: abstract syntax, concrete syntax and semantics.

The next three subsections describe the abstract, concrete syntax and semantics of Ann.

4.1 Abstract syntax

The abstract syntax describes the structure of the language and the way its different elements can be combined. It has been specified in Ann by using a meta-model[1], which can be found simplified in Figure 2.

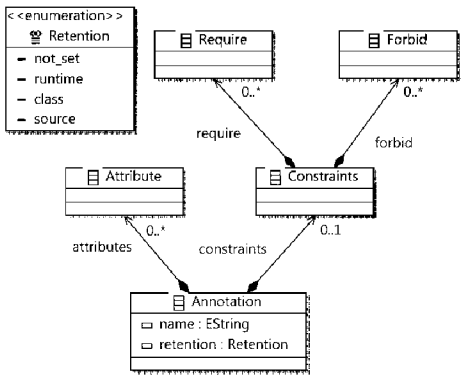

Figure 2: Simplified meta-model excerpt representing the abstract syntax of Ann.

The `Annotation` meta-class contains both the attributes of an annotation and its associated constraints. Details concerning attributes have been omitted; and constraints are split into two types: requirements (class `Require`) and prohibitions (class `Forbid`). In Figure 3 we can see an expanded section of this meta-model, in particular the one concerning the constraints.

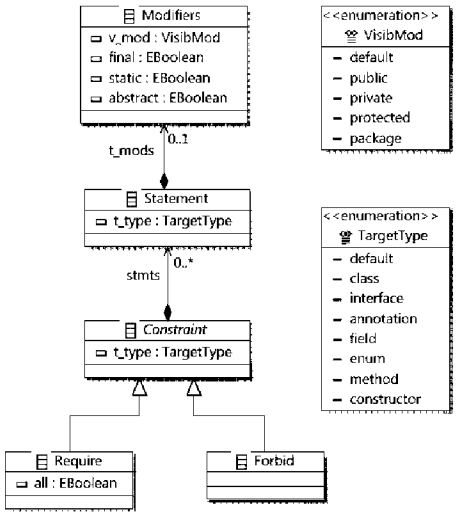

Figure 3: Meta-model excerpt for annotation constraints.

Each *statement* represents a description of a Java element (like `class`, `interface` or `field`) over which the annotation is (dis-)allowed. Several statements are possible within the

[1]A meta-model is a model which describes the properties of a set of models, and so it is in a higher level of abstraction

same constraint (e.g., if the same annotation can be applied to several targets), enhancing the expressive power of Ann. There is also the possibility of expressing constraints for specific target types (e.g., a `field`), which indicates that the given constraint only applies when the annotation is attached to that target type (e.g., a `field`). An annotation is correctly placed at a target type if it satisfies some of the statements of the positive requirements for the given target, and none of the prohibitions.

It will be shown in Section 6 that these two types of constraints, and their combinations have enough expressive power to cover a huge scope of the full conceptual model of an annotation group design in a real use case.

4.2 Concrete syntax

The concrete syntax of a DSL describes the specific representation of the language, and hence how users visualize or create models. Concrete syntaxes can be textual or graphical. Given that one of the goals of Ann is to give a friendlier syntax for Java developers defining annotations, mitigating the incoherences that can be found nowadays in Java language, a textual concrete syntax has been chosen for it.

An excerpt of the concrete syntax definition for the constraints within an annotation can be found in Listing 3, represented in Extended Backus-Naur Form.

```
1 <Forbid> ::=
2    "forbid" <Statement> ("and" <Statement>)* ";" |
3   "at" <TargetType> ":"
4       "forbid" <Statement> ("and" <Statement>)* ";";
5
6 <Require> ::=
7    "require" <Statement> ("or" <Statement>)* ";" |
8    "at" <TargetType> ":"
9       "require" "all"? <Statement> ("or" <Statement>)* ";";
```

Listing 3: Concrete syntax excerpt for constraints in Ann.

Listing 4 shows how the Java annotation type `Person` previously shown in Listing 1 would be described using Ann. A new keyword (`annotation`) is used on its declaration (line 3), and the overall result is a clearer, more readable code. The definition and initialisation of the attributes is now in harmony with the usual Java syntax for the same purpose.

```
1  package examples;
2
3  annotation Person {
4     String name = "Mary";
5     int age = 21;
6     float weight = 52.3;
7
8     require public class; // annotation allowed for classes...
9
10    at class: forbid final field; // ... with no final fields
11 }
```

Listing 4: Annotation `Person` defined in Ann.

Regarding the restriction of the allowed targets, we can now express some more elaborated descriptions, in this case that Person can only annotate public classes (line 8) with no final fields (line 10). We recall that with Java the closer we got to this statement was that the annotation could have as targets classes, interfaces and enumerations, which is much more general than we intend.

In the concrete syntax for requirements, we note also the special keyword `all`. This would apply if, for instance, we would want that all the methods of the classes annotated with `Person` were also public. Then we would add the clause `at class: require all public method.`

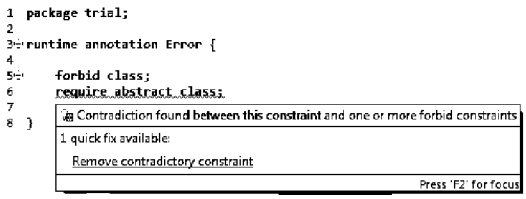

Figure 4: Validation of contradictory constraints.

4.3 Semantics: code generation

The semantics of a modelling language can be specified by several means, like e.g., providing an interpreter or a code generator. In the present case, code generation has been the adopted solution.

In order to fully specify the semantics of Ann, it is necessary to generate on one hand the Java code associated with the definition of the annotations; and on the other the code of the processors. The latter will ensure that the constraints specified for each of the defined annotations are being met.

For each of the annotations defined at most two processors will be generated, one for checking the requirements and the other for checking the prohibitions. The structure of the annotation processors generated complies with the one presented in Section 2: each of the relevant elements of the Java program is looked up to check whether its properties satisfy the specified requirements or prohibitions.

5. TOOL SUPPORT

Ann has been developed using the Eclipse Modelling Framework (EMF) [16]. Integrated in this environment, different tools have been used for the different elements of the DSL. The meta-model has been described using the meta-modelling language Ecore, which is based in a subset of UML class diagrams for the description of structural aspects.

Xtext [7] has been used to define the textual concrete syntax. Xtext is integrated with EMF and able to generate a fully customisable and complete editor for the defined language. In our present case we have added the validation of, among other issues, contradictory constraints specified within an annotation, providing relevant quick fixes, as can be seen in Figure 4.

Finally, the code generator has been developed using the language Xtend, included in the framework Xtext. Xtend is a Java dialect more expressive and flexible, with facilities for model navigation. It also allows creating generation templates, what makes it specially useful for code generation.

The result is an Eclipse plug-in, which is seamlessly integrated within the Eclipse Java Development Tools (JDT).

6. A REAL USE CASE: JPA ANNOTATIONS

A subset of JPA annotations has been chosen in order to test the Ann DSL: `Entity`, `Id`, `IdClass`, `Embeddable` and `EmbeddedId`.

This selection has been made according to their extensive use in the JPA context, given that all of them are used to describe entities and their primary keys, central concepts in database design.

6.1 Defining the annotations with Ann

The constraints associated with the `Entity` annotation were outlined in Section 2. Moreover, given that it defines an entity within a database, a corresponding primary key must also be specified. The other selected annotations are used precisely for this purpose.

An entity may have a simple or compound primary key. For the former case, the annotation `Id` is used; for the latter, the annotations `IdClass` or `EmbeddedId` and `Embeddable`[2].

```
1  runtime annotation Entity {
2     String name = "";
3
4     require class;
5     forbid final class;
6
7     at class: require public constructor or protected constructor;
8     at class: forbid final method;
9
10    at class: require @Id method or @Id field or
11                  @EmbeddedId method or @EmbeddedId field;
12    at class: forbid @Id method and @EmbeddedId method;
13    at class: forbid @Id field and @EmbeddedId field;
14 }
15
16 runtime annotation Embeddable {
17    require class;
18
19    at class: forbid @Id method;
20    at class: forbid @EmbeddedId method;
21
22    at class: forbid @Id field;
23    at class: forbid @EmbeddedId field;
24 }
25
26 runtime annotation EmbeddedId {
27    require method or field;
28
29    at field: require @Entity class;
30    at method: require @Entity class;
31 }
32
33 runtime annotation Id {
34    require method or field;
35
36    at field: require @Entity class;
37    at method: require @Entity class;
38 }
39
40 runtime annotation IdClass {
41    Class value;
42
43    require @Entity class;
44 }
```

Listing 5: Selected JPA annotations defined in Ann.

Listing 5 shows the description of the explained annotations using Ann. Clearly the chosen subset of annotations is very interrelated given all the respective constraints that can be noticed. For example, a class annotated with `Embeddable` (lines 16-24) acts as a primary key for another class, in which it is embedded, and thus it must not have a primary key itself, prohibition which is expressed through lines 19-23.

Alternatively, the annotation `IdClass` (lines 40-44) can be used to specify the class that contains the fields which form the compound primary key. Therefore it can only be attached to classes annotated with `Entity`, requirement described in line 43.

Annotations `Id` (lines 33-38) and `EmbeddedId` (lines 25-30) mark the primary key of an entity, and thus can only annotate methods or fields (lines 34 and 27 resp.) which form part of a class annotated with `Entity` (lines 36-37 and 29-30 resp.).

Finally, regarding the `Entity` annotation (lines 1-14), structural properties of the annotated classes are expressed throughout lines 4-8; and lines 10-11 establish the need of a primary key through a requirement, among other constraints.

After the definition of all the annotations and their constraints, the corresponding code is generated and ready to use in both new or existing Java projects.

[2]Depending on whether using fields or an embeddable class to represent the compound key, respectively.

```
1  package test;
2
3  import jpa.Entity;
5
6  @Entity
7  public class Persona {
8      private I
9
10     public Pe
11     }
12 }
```

The annotation @Entity is disallowed for this location. @Id method OR @Id field OR @EmbeddedId method OR @EmbeddedId field required as member.

Press 'F2' for focus

Figure 5: Entity without primary key.

```
1  package test;
2
3  import jpa.Entity;
5
6  //@Entity
7  public class Persona {
8      @Id private int DNI;
9
10     public Persona(){
11     }
12 }
```

The annotation @Id is disallowed for this location. @Entity class required as container.

Press 'F2' for focus

Figure 6: Primary key in a field not belonging to an entity.

6.2 Using the generated code

The generated processors are capable of detecting where a constraint is being violated and also notify the developer by means of an explanatory message.

In Figure 5 the annotation `Entity` is being used on a class and no primary key is being specified, situation not allowed in the JPA context.

Another example of misuse is the one shown in Figure 6. In this case, the annotation `Id` is used in a field inside a class that is not annotated as `Entity`, situation that leads to another error.

7. RELATED RESEARCH

Some research has been made in order to improve and expand the functionality of Java annotations. For example, Phillips in [14] aims at conciliating object oriented principles with the design of annotations by the introduction of a new one: `composite`. With it, he manages to support composition, allowing encapsulation and polymorphism of annotations.

A Java extension, `@Java`, is proposed by Cazzola and Vacchi [10] in order to expand the range of application of an annotation to code blocks and expressions, although some improvement in this respect has also been made natively in the latest version of Java [6].

The expressiveness limitations of Java annotations are recognised in [11], where a proposal is made to embed DSLs into Java, with a more natural and flexible syntax. JUMP [8] is a tool to reverse engineer Java programs (with annotations) into profiled UML class diagrams.

Although the aforementioned approaches expand the features of Java annotations, they do not address the integrity and design issues explained throughout this paper, which is the main goal of our work.

Just a few works are aimed at improving the design of

annotations. Darwin [12] suggests a DSL, called AnnaBot, based on *claims* about a set of existing annotations, with a concrete syntax very similar to Java. With this *claims* interdependency constraints can be expressed within a set of annotations. However, there is no possibility of characterising the targets of an annotation type. Moreover, no improvement is made with respect to the syntax for defining annotations in Java, given its heavy focus on existing sets of annotations and constraints between them, and not on isolated ones. Finally, the approach uses reflection to check the statements of its *claims*, which could and should be avoided.

Another approach is AVal [13], a set of meta-annotations[3] to add integrity constraints at the definition of the annotation type. This approach has as a drawback that its expressive possibilities are rather restricted, given the limited flexibility which meta-annotations provide. For example, a simple constraint we saw for entities in Section 2 was that no class methods should be final, and this cannot be expressed by the meta-annotations provided in AVal.

Hence, to the best of our knowledge, Ann is the first proposal for a high-level means to describe both the syntax and well-formedness constraints of annotation families, making explicit the design of such annotation set and allowing their immediate use on Java projects thanks to the code generation facility.

8. CONCLUSIONS AND FUTURE WORK

Ann makes possible the effective design of Java annotations by improving their native syntactical support and allowing the expression of integrity constraints both related to an annotation type and within a set of annotations. Thanks to the code generator, the approach can be perfectly integrated with existing Java projects. Moreover, with the use of annotation processors all the integrity constraints described with the DSL are checked at compile time, which improves both usability and efficiency. This is because it is not necessary to execute the application in order to know whether the annotations are being correctly used, hence saving much time and effort for developers.

Concerning future work, a huge range of possibilities is available given the flexibility that a DSL provides. As seen in Section 7, the meta-model of Java annotations can be still improved and expanded to improve its harmony with the rest of Java elements, like, for example, its conciliation with object-oriented principles such as composition, inheritance and polymorphism, which might help to make cleaner the design of a set of annotations.

At present two basic types of constraints are considered in Ann (requirements and prohibitions), which are enough to express common integrity constraints as it has been seen in Section 6. However, further experimentation could reveal new constraint types or combinations, which could be added to the DSL in the future, given the flexibility that a meta-model provides. Another line of work is the reverse engineering of annotation constraints from the analysis of annotated Java programs.

Acknowledgements. This work was supported by the Spanish Ministry of Economy and Competitivity with project Go-Lite (TIN2011-24139) and the Community of Madrid with project SICOMORO (S2013/ICE-3006).

[3]Annotations whose targets are other annotations.

9. REFERENCES

[1] Hibernate ORM. http://hibernate.org/orm/. Accessed: 2014-09-20.

[2] Java annotations in 5th edition. http://docs.oracle.com/javase/1.5.0/docs/guide/language/annotations.html. Accessed: 2014-07-13.

[3] Java Enterprise Edition 7 API. http://docs.oracle.com/javaee/7/api/. Accessed: 2014-07-13.

[4] Seam annotations. http://docs.jboss.org/seam/latest/reference/html/annotations.html. Accessed: 2014-09-20.

[5] Spring. http://spring.io/. Accessed: 2014-09-20.

[6] Type Annotations and Pluggable Type Systems. http://docs.oracle.com/javase/tutorial/java/annotations/type_annotations.html. Accessed: 2014-08-31.

[7] Xtext. http://www.eclipse.org/Xtext/. Accessed: 2014-09-20.

[8] Bergmayr, A., Grossniklaus, M., Wimmer, M., and Kappel, G. JUMP - From Java Annotations to UML Profiles. In *17th International Conference on Model Driven Engineering Languages and Systems, MODELS'14* (2014), p. to appear.

[9] Brambilla, M., Cabot, J., and Wimmer, M. *Model-Driven Software Engineering in Practice.* Synthesis Lectures on Software Engineering. Morgan & Claypool, USA, 2012.

[10] Cazzola, W., and Vacchi, E. @java: Bringing a richer annotation model to java. *Computer Languages, Systems & Structures 40*, 1 (2014), 2 – 18. Special issue on the Programming Languages track at the 28th ACM Symposium on Applied Computing.

[11] Clark, T., Sammut, P., and Willans, J. S. Beyond annotations: A proposal for extensible java (XJ). In *Eighth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2008), 28-29 September 2008, Beijing, China* (2008), pp. 229–238.

[12] Darwin, I. Annabot: A static verifier for java annotation usage. *Advances in Software Engineering* (2010).

[13] Noguera, C., and Pawlak, R. AVal: an Extensible Attribute-Oriented Programming Validator for Java. In *6th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'06)* (Philadelphia, PA, USA, 2006), IEEE, pp. 175–183.

[14] Phillips, A. @composite: Macro annotations for java c. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications* (New York, NY, USA, 2009), OOPSLA '09, ACM, pp. 767–768.

[15] Schildt, H. *Java: The complete reference, J2SE*, 5th ed. McGraw-Hill, New York, 2005.

[16] Steinberg, D., Budinsky, F., Paternostro, M., and Merks, E. *EMF: Eclipse Modeling Framework, 2nd Edition.* Addison-Wesley Professional, 2008. See also http://www.eclipse.org/modeling/emf/.

[17] Voelter, M. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages.* dslbook.org, 2013.